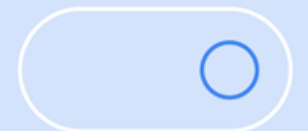


**WELCOME !**

# Front-end Development: JS

Benjamin Ogbonna  
Abocoders



# OUTLINE

- Introduction
- Output
- Statements
- syntax
- Comments
- Variables
- Operators
- Arithmetic
- Assignment
- Data Type
- Functions
- Objects
- Events
- String, String Methods
- Numbers, Number Methods
- Arrays, Array Methods
- Array Sort
- Array Iteration
- Dates, Date Formats
- Date Get Methods
- Math
- Comparisons
- Conditions
- Loops: for & while loops

# INTRODUCTION

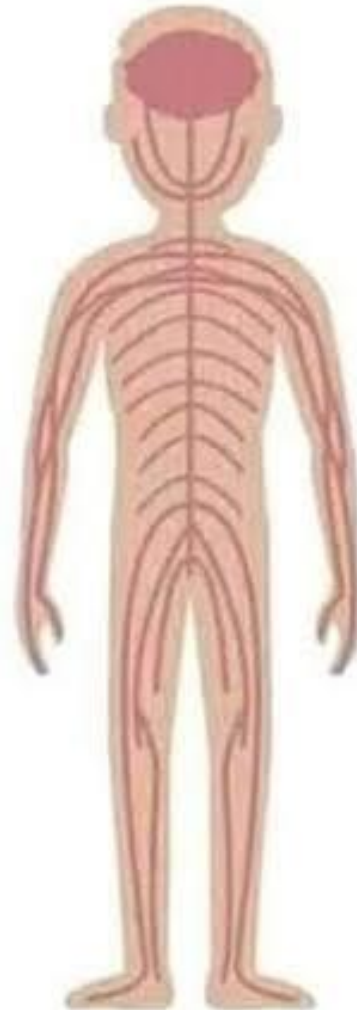
JS

# ANALOGY

HTML

JS

CSS



# JavaScript Variables

JavaScript variables are containers for storing data values.

In this example, `x`, `y`, and `z`, are variables:

## Example

```
var x = 5;  
var y = 6;  
var z = x + y;
```

[Try it Yourself »](#)

# Much Like Algebra

In this example, `price1`, `price2`, and `total`, are variables:

## Example

```
var price1 = 5;  
var price2 = 6;  
var total = price1 + price2;
```

[Try it Yourself »](#)

# JavaScript Identifiers

All JavaScript **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y), or more descriptive names (age, sum, totalVolume).

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs.
- Names must begin with a letter
- Names can also begin with \$ and \_ (but we will not use it in this tutorial)
- Names are case sensitive (y and Y are different variables)
- Reserved words (like JavaScript keywords) cannot be used as names



JavaScript identifiers are case-sensitive.



# JavaScript Data Types

JavaScript variables can hold numbers like 100, and text values like "John Doe".

In programming, text values are called text strings.

JavaScript can handle many types of data, but for now, just think of numbers and strings.

Strings are written inside double or single quotes. Numbers are written without quotes.

If you put quotes around a number, it will be treated as a text string.

## Example

```
var pi = 3.14;  
var person = "John Doe";  
var answer = 'Yes I am!';
```

Try it Yourself »

# Declaring (Creating) JavaScript Variables

Creating a variable in JavaScript is called "declaring" a variable.

You declare a JavaScript variable with the **var** keyword:

```
var carName;
```

After the declaration, the variable has no value. (Technically it has the value of **undefined**)

To **assign** a value to the variable, use the equal sign:

```
carName = "Volvo";
```

You can also assign a value to the variable when you declare it:

```
var carName = "Volvo";
```

# Value = undefined

In computer programs, variables are often declared without a value. The value can be something that has to be calculated, or something that will be provided later, like user input.

A variable declared without a value will have the value **undefined**.

The variable `carName` will have the value `undefined` after the execution of this statement:

## Example

```
var carName;
```

Try it yourself »

# JavaScript Arithmetic

As with algebra, you can do arithmetic with JavaScript variables, using operators like `=` and `+`:

## Example

```
var x = 5 + 2 + 3;
```

[Try it yourself »](#)

You can also add strings, but strings will be concatenated (added end-to-end):

## Example

```
var x = "John" + " " + "Doe";
```

[Try it Yourself »](#)



# Operators

An operator is a symbol that tells the compiler or interpreter to perform specific mathematical, relational or logical operations.

# JavaScript Arithmetic Operators

Arithmetic operators are used to perform arithmetic on numbers (literals or variables).

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement

# Adding Strings and Numbers

Adding two numbers, will return the sum, but adding a number and a string will return a string:

## Example

```
x = 5 + 5;  
y = "5" + 5;  
z = "Hello" + 5;
```

The result of x, y, and z will be:

# JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

Operator	Example	Same As
=	<code>x = y</code>	<code>x = y</code>
+=	<code>x += y</code>	<code>x = x + y</code>
-=	<code>x -= y</code>	<code>x = x - y</code>
*=	<code>x *= y</code>	<code>x = x * y</code>
/=	<code>x /= y</code>	<code>x = x / y</code>
%=	<code>x %= y</code>	<code>x = x % y</code>



# JavaScript Comparison and Logical Operators

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

# Arithmetic Operations

A typical arithmetic operation operates on two numbers.

The two numbers can be literals:

## Example

```
var x = 100 + 50;
```

[Try it Yourself »](#)

or variables:

## Example

```
var x = a + b;
```

[Try it Yourself »](#)

or expressions:

## Example

```
var x = (100 + 50) * a;
```

[Try it Yourself »](#)

# Operators and Operands

The numbers (in an arithmetic operation) are called **operands**.

The operation (to be performed between the two operands) is defined by an **operator**.

Operand	Operator	Operand
100	+	50

The **addition** operator (+) adds numbers:

# Operator Precedence

Operator precedence describes the order in which operations are performed in an arithmetic expression.

## Example

```
var x = 100 + 50 * 3;
```

Try it Yourself »

Is the result of example above the same as  $150 * 3$ , or is it the same as  $100 + 150$ ?

Is the addition or the multiplication done first?

As in traditional school mathematics, the multiplication is done first.

Multiplication (\*) and division (/) have higher **precedence** than addition (+) and subtraction (-).

And (as in school mathematics) the precedence can be changed by using parentheses:



## Example

```
var x = (100 + 50) * 3;
```

[Try it Yourself »](#)

When using parentheses, the operations inside the parentheses are computed first.

When many operations have the same precedence (like addition and subtraction), they are computed from left to right:

## Example

```
var x = 100 + 50 - 3;
```

[Try it Yourself »](#)

# JavaScript Operator Precedence Values

Value	Operator	Description	Example
19	( )	Expression grouping	(3 + 4)
18	.	Member	person.name
18	[]	Member	person["name"]
17	()	Function call	myFunction()
17	new	Create	new Date()

16	++	Postfix Increment	i++
16	--	Postfix Decrement	i--
15	++	Prefix Increment	++i
15	--	Prefix Decrement	--i
15	!	Logical not	!(x==y)
15	typeof	Type	typeof x
14	*	Multiplication	10 * 5
14	/	Division	10 / 5
14	%	Modulo division	10 % 5
14	**	Exponentiation	10 ** 2
13	+	Addition	10 + 5
13	-	Subtraction	10 - 5





The = assignment operator assigns a value to a variable.

## Assignment

```
var x = 10;
```

Try it Yourself »

The += assignment operator adds a value to a variable.

## Assignment

```
var x = 10;
```

```
x += 5;
```

Try it Yourself »

- The -= assignment operator subtracts a value from a variable.

# JavaScript Data Types

JavaScript variables can hold many **data types**: numbers, strings, arrays, objects and more:

```
var length = 16;           // Number
var lastName = "Johnson"; // String
var cars = ["Saab", "Volvo", "BMW"]; // Array
var x = {firstName:"John", lastName:"Doe"}; // Object
```

## The Concept of Data Types

In programming, data types is an important concept.

To be able to operate on variables, it is important to know something about the type.

Without data types, a computer cannot safely solve this:

```
var x = 16 + "Volvo";
```

# JavaScript Has Dynamic Types

JavaScript has dynamic types. This means that the same variable can be used as different types:

## Example

```
var x;           // Now x is undefined
var x = 5;       // Now x is a Number
var x = "John";  // Now x is a String
```

# JavaScript Strings

A string (or a text string) is a series of characters like "John Doe".

Strings are written with quotes. You can use single or double quotes:

## Example

```
var carName = "Volvo XC60";    // Using double quotes
var carName = 'Volvo XC60';    // Using single quotes
```

You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

## Example

```
var answer = "It's alright";    // Single quote inside double quotes
var answer = "He is called 'Johnny'"; // Single quotes inside double quotes
var answer = 'He is called "Johnny"'; // Double quotes inside single quotes
```

# JavaScript Numbers

JavaScript has only one type of numbers.

Numbers can be written with, or without decimals:

## Example

```
var x1 = 34.00;    // Written with decimals
var x2 = 34;       // Written without decimals
```

Extra large or extra small numbers can be written with scientific (exponential) notation:

## Example

```
var y = 123e5;     // 12300000
var z = 123e-5;    // 0.00123
```

# JavaScript Booleans

Booleans can only have two values: true or false.

## Example

```
var x = true;  
var y = false;
```

Booleans are often used in conditional testing.

You will learn more about conditional testing later in this tutorial.

# JavaScript Arrays

JavaScript arrays are written with square brackets.

Array items are separated by commas.

The following code declares (creates) an array called cars, containing three items (car names):

## Example

```
var cars = ["Saab", "Volvo", "BMW"];
```

Try it Yourself »

Array indexes are zero-based, which means the first item is [0], second is [1], and so on.

# JavaScript Objects

JavaScript objects are written with curly braces.

Object properties are written as name:value pairs, separated by commas.

## Example

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

Try it Yourself »

The object (person) in the example above has 4 properties: firstName, lastName, age, and eyeColor.



# The typeof Operator

You can use the JavaScript **typeof** operator to find the type of a JavaScript variable:

## Example

```
typeof "John"           // Returns string
typeof 3.14              // Returns number
typeof false            // Returns boolean
typeof [1,2,3,4]         // Returns object
typeof {name:'John', age:34} // Returns object
```

Try it Yourself »



In JavaScript, an array is a special type of object. Therefore `typeof [1,2,3,4]` returns object.



# Undefined

In JavaScript, a variable without a value, has the value **undefined**. The typeof is also **undefined**.

## Example

```
var person; // Value is undefined, type is undefined
```

Try it Yourself »

Any variable can be emptied, by setting the value to **undefined**. The type will also be **undefined**.

## Example

```
person = undefined; // Value is undefined, type is undefined
```

Try it Yourself »

# Empty Values

An empty value has nothing to do with undefined.

An empty string variable has both a value and a type.

## Example

```
var car = ""; // The value is "", the typeof is string
```

Try it Yourself »

# Null

In JavaScript null is "nothing". It is supposed to be something that doesn't exist.

Unfortunately, in JavaScript, the data type of null is an object.



You can consider it a bug in JavaScript that `typeof null` is an object. It should be null.

You can empty an object by setting it to null:

## Example

```
var person = null;           // Value is null, but type is still an object
```

Try it Yourself »

# JavaScript Functions

[« Previous](#)

[Next Chapter »](#)

A JavaScript function is a block of code designed to perform a particular task.

A JavaScript function is executed when "something" invokes it (calls it).

## Example

```
function myFunction(p1, p2) {  
  return p1 * p2;           // The function returns the product of p1 and p2  
}
```

[Try it yourself »](#)

# JavaScript Function Syntax

A JavaScript function is defined with the **function** keyword, followed by a **name**, followed by parentheses **()**.

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas: **(parameter1, parameter2, ...)**

The code to be executed, by the function, is placed inside curly brackets: **{ }**

```
function name(parameter1, parameter2, parameter3) {  
    code to be executed  
}
```

Function **parameters** are the **names** listed in the function definition.

Function **arguments** are the real **values** received by the function when it is invoked.

Inside the function, the arguments behave as local variables.



A Function is much the same as a Procedure or a Subroutine, in other programming languages.

# Function Invocation

The code inside the function will execute when "something" **invokes** (calls) the function:

- When an event occurs (when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self invoked)

You will learn a lot more about function invocation later in this tutorial.

---

# Function Return

When JavaScript reaches a **return statement**, the function will stop executing.

If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.

Functions often compute a **return value**. The return value is "returned" back to the "caller":

## Example

Calculate the product of two numbers, and return the result:

```
var x = myFunction(4, 3);      // Function is called, return value will end up in x

function myFunction(a, b) {
  return a * b;                // Function returns the product of a and b
}
```

The result in x will be:

12

[Try it yourself »](#)



# Why Functions?

You can reuse code: Define the code once, and use it many times.

You can use the same code many times with different arguments, to produce different results.

## Example

Convert Fahrenheit to Celsius:

```
function toCelsius(fahrenheit) {  
    return (5/9) * (fahrenheit-32);  
}  
document.getElementById("demo").innerHTML = toCelsius(77);
```

[Try it yourself »](#)

# Functions Used as Variables

In JavaScript, you can use functions the same way as you use variables.

## Example

You can use:

```
var text = "The temperature is " + toCelsius(77) + " Celsius";
```

Instead of:

```
var x = toCelsius(32);  
var text = "The temperature is " + x + " Celsius";
```

[Try it Yourself »](#)

# JavaScript Objects

You have already learned that JavaScript variables are containers for data values.

This code assigns a **simple value** (Fiat) to a **variable** named car:

```
var car = "Fiat";
```

Try it Yourself »

Objects are variables too. But objects can contain many values.

This code assigns **many values** (Fiat, 500, white) to a **variable** named car:

```
var car = {type:"Fiat", model:"500", color:"white"};
```

Try it Yourself »

The values are written as **name:value** pairs (name and value separated by a colon).



JavaScript objects are containers for **named values**.

# Object Properties

The name:values pairs (in JavaScript objects) are called **properties**.

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue

# Object Definition

You define (and create) a JavaScript object with an object literal:

## Example

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

[Try it Yourself »](#)

Spaces and line breaks are not important. An object definition can span multiple lines:

## Example

```
var person = {  
  firstName:"John",  
  lastName:"Doe",  
  age:50,  
  eyeColor:"blue"  
};
```

# Accessing Object Properties

You can access object properties in two ways:

```
objectName.propertyName
```

or

```
objectName["propertyName"]
```

## Example1

```
person.lastName;
```

## Example2

```
person["lastName"];
```

# Do Not Declare Strings, Numbers, and Booleans as Objects!

When a JavaScript variable is declared with the keyword "new", the variable is created as an object:

```
var x = new String();      // Declares x as a String object
var y = new Number();      // Declares y as a Number object
var z = new Boolean();     // Declares z as a Boolean object
```

Avoid String, Number, and Boolean objects. They complicate your code and slow down execution speed.

# JavaScript Events

[« Previous](#)

[Next](#)

HTML events are "**things**" that happen to HTML elements.

When JavaScript is used in HTML pages, JavaScript can "**react**" on these events.

## HTML Events

An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

Often, when events happen, you may want to do something.

JavaScript lets you execute code when events are detected.



HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.

With single quotes:

```
<some-HTML-element some-event='some JavaScript'>
```

With double quotes:

```
<some-HTML-element some-event="some JavaScript">
```

In the following example, an onclick attribute (with code), is added to a button element:

## Example

```
<button onclick='getElementById("demo").innerHTML=Date()'>The time is?</button>
```

Try it Yourself »

In the example above, the JavaScript code changes the content of the element with id="demo".

In the next example, the code changes the content of its own element (using **this.innerHTML**):

## Example

```
<button onclick="this.innerHTML=Date()">The time is?</button>
```

Try it Yourself »



JavaScript code is often several lines long. It is more common to see event attributes calling functions:



# Common HTML Events

Here is a list of some common HTML events:

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

The list is much longer: [W3Schools JavaScript Reference HTML DOM Events](#).

# What can JavaScript Do?

Event handlers can be used to handle, and verify, user input, user actions, and browser actions:

- Things that should be done every time a page loads
- Things that should be done when the page is closed
- Action that should be performed when a user clicks a button
- Content that should be verified when a user inputs data
- And more ...

Many different methods can be used to let JavaScript work with events:

- HTML event attributes can execute JavaScript code directly
- HTML event attributes can call JavaScript functions
- You can assign your own event handler functions to HTML elements
- You can prevent events from being sent or being handled
- And more ...

# JavaScript Strings

[« Previous](#)

[Next Chapter »](#)

JavaScript strings are used for storing and manipulating text.

## JavaScript Strings

A JavaScript string simply stores a series of characters like "John Doe".

A string can be any text inside quotes. You can use single or double quotes:

### Example

```
var carname = "Volvo XC60";  
var carname = 'Volvo XC60';
```

[Try it Yourself »](#)

You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

## Example

```
var answer = "It's alright";  
var answer = "He is called 'Johnny'";  
var answer = 'He is called "Johnny"';
```

[Try it Yourself »](#)



Developer Student Clubs

Google Developers

# String Length

The `length` of a string is found in the built in property **`length`**:

## Example

```
var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";  
var sln = txt.length;
```

[Try it Yourself »](#)

# Special Characters

Because strings must be written within quotes, JavaScript will misunderstand this string:

```
var y = "We are the so-called "Vikings" from the north."
```

The string will be chopped to "We are the so-called ".

The solution to avoid this problem, is to use the **\ escape character**.

The backslash escape character turns special characters into string characters:

## Example

```
var x = 'It\'s alright';  
var y = "We are the so-called \"Vikings\" from the north."
```

Try it Yourself »





The escape character (\) can also be used to insert other special characters in a string.

This is the list of special characters that can be added to a text string with the backslash sign:

Code	Outputs
\'	single quote
\"	double quote
\\	backslash
\n	new line
\r	carriage return
\t	tab
\b	backspace
\f	form feed

# Breaking Long Code Lines

For best readability, programmers often like to avoid code lines longer than 80 characters.

If a JavaScript statement does not fit on one line, the best place to break it is after an operator:

## Example

```
document.getElementById("demo").innerHTML =  
"Hello Dolly.";
```

Try it Yourself »

You can also break up a code line **within a text string** with a single backslash:

## Example

```
document.getElementById("demo").innerHTML = "Hello \  
Dolly!";
```

# String Methods

Method	Description
charAt()	Returns the character at the specified index (position)
charCodeAt()	Returns the Unicode of the character at the specified index
concat()	Joins two or more strings, and returns a copy of the joined strings
fromCharCode()	Converts Unicode values to characters
indexOf()	Returns the position of the first found occurrence of a specified value in a string
lastIndexOf()	Returns the position of the last found occurrence of a specified value in a string
localeCompare()	Compares two strings in the current locale
match()	Searches a string for a match against a regular expression, and returns the matches
replace()	Searches a string for a value and returns a new string with the value replaced
search()	Searches a string for a value and returns the position of the match

<code>slice()</code>	Extracts a part of a string and returns a new string
<code>split()</code>	Splits a string into an array of substrings
<code>substr()</code>	Extracts a part of a string from a start position through a number of characters
<code>substring()</code>	Extracts a part of a string between two specified positions
<code>toLocaleLowerCase()</code>	Converts a string to lowercase letters, according to the host's locale
<code>toLocaleUpperCase()</code>	Converts a string to uppercase letters, according to the host's locale
<code>toLowerCase()</code>	Converts a string to lowercase letters
<code>toString()</code>	Returns the value of a String object
<code>toUpperCase()</code>	Converts a string to uppercase letters
<code>trim()</code>	Removes whitespace from both ends of a string
<code>valueOf()</code>	Returns the primitive value of a String object

# JavaScript String Methods

[« Previous](#)

[Next Chapter »](#)

String methods help you to work with strings.

## Finding a String in a String

The **indexOf()** method returns the index of (the position of) the **first** occurrence of a specified text in a string:

### Example

```
var str = "Please locate where 'locate' occurs!";  
var pos = str.indexOf("locate");
```

[Try it Yourself »](#)

# Searching for a String in a String

The **search()** method searches a string for a specified value and returns the position of the match:

## Example

```
var str = "Please locate where 'locate' occurs!";  
var pos = str.search("locate");
```

Try it Yourself »



## Did You Notice?

The two methods, `indexOf()` and `search()`, are equal.

They accept the same arguments (parameters), and they return the same value.

The two methods are equal, but the `search()` method can take much more powerful search values.

# Extracting String Parts

There are 3 methods for extracting a part of a string:

- `slice(start, end)`
- `substring(start, end)`
- `substr(start, length)`

## The `slice()` Method

**`slice()`** extracts a part of a string and returns the extracted part in a new string.

The method takes 2 parameters: the starting index (position), and the ending index (position).

This example slices out a portion of a string from position 7 to position 13:

### Example

```
var str = "Apple, Banana, Kiwi";  
var res = str.slice(7,13);
```

The result of `res` will be:

```
Banana
```

If a parameter is negative, the position is counted from the end of the string.

This example slices out a portion of a string from position -12 to position -6:

## Example

```
var str = "Apple, Banana, Kiwi";  
var res = str.slice(-12, -6);
```

The result of res will be:

Banana

[Try it Yourself »](#)

If you omit the second parameter, the method will slice out the rest of the string:

## Example

```
var res = str.slice(7);
```



# The substring() Method

**substring()** is similar to `slice()`.

The difference is that `substring()` cannot accept negative indexes.

## Example

```
var str = "Apple, Banana, Kiwi";  
var res = str.substring(7,13);
```

The result of *res* will be:

Banana

[Try it yourself »](#)

If you omit the second parameter, `substring()` will slice out the rest of the string.

# Replacing String Content

The **replace()** method replaces a specified value with another value in a string:

## Example

```
str = "Please visit Microsoft!";  
var n = str.replace("Microsoft","W3Schools");
```

Try it Yourself »

The replace() method can also take a regular expression as the search value.

By default, the replace() function replaces only the first match. To replace all matches, use a regular expression with a g flag (for global match):

## Example

```
str = "Please visit Microsoft!";  
var n = str.replace(/Microsoft/g,"W3Schools");
```

# Converting to Upper and Lower Case

A string is converted to upper case with **toUpperCase()**:

## Example

```
var text1 = "Hello World!";    // String
var text2 = text1.toUpperCase(); // text2 is text1 converted to upper
```

Try it Yourself »

A string is converted to lower case with **toLowerCase()**:

## Example

```
var text1 = "Hello World!";    // String
var text2 = text1.toLowerCase(); // text2 is text1 converted to lower
```

# The concat() Method

**concat()** joins two or more strings:

## Example

```
var text1 = "Hello";  
var text2 = "World";  
text3 = text1.concat(" ",text2);
```

Try it Yourself »

The **concat()** method can be used instead of the plus operator. These two lines do the same:

## Example

```
var text = "Hello" + " " + "World!";  
var text = "Hello".concat(" ", "World!");
```



All string methods return a new string. They don't modify the original string.  
Formally said: Strings are immutable: Strings cannot be changed, only replaced.

# Extracting String Characters

There are 2 **safe** methods for extracting string characters:

- `charAt(position)`
  - `charCodeAt(position)`
- 

## The `charAt()` Method

The **`charAt()`** method returns the character at a specified index (position) in a string:

### Example

```
var str = "HELLO WORLD";  
str.charAt(0);           // returns H
```

# JavaScript Numbers

[« Previous](#)

[Next Chapter »](#)

JavaScript has only one type of number.

Numbers can be written with, or without, decimals.

## JavaScript Numbers

JavaScript numbers can be written with, or without decimals:

### Example

```
var x = 34.00;    // A number with decimals
var y = 34;       // A number without decimals
```

# JavaScript Numbers are Always 64-bit Floating Point

Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point etc.

JavaScript numbers are always stored as double precision floating point numbers, following the international IEEE 754 standard.

This format stores numbers in 64 bits, where the number (the fraction) is stored in bits 0 to 51, the exponent in bits 52 to 62, and the sign in bit 63:

Value (aka Fraction/Mantissa)	Exponent	Sign
52 bits (0 - 51)	11 bits (52 - 62)	1 bit (63)

# Hexadecimal

JavaScript interprets numeric constants as hexadecimal if they are preceded by 0x.

## Example

```
var x = 0xFF;           // x will be 255
```

Try it Yourself »



Never write a number with a leading zero (like 07).

Some JavaScript versions interpret numbers as octal if they are written with a leading zero.



# NaN - Not a Number

NaN is a JavaScript reserved word indicating that a value is not a number.

Trying to do arithmetic with a non-numeric string will result in NaN (Not a Number):

## Example

```
var x = 100 / "Apple"; // x will be NaN (Not a Number)
```

[Try it Yourself »](#)

However, if the string contains a numeric value , the result will be a number:

## Example

```
var x = 100 / "10"; // x will be 10
```

[Try it Yourself »](#)

# JavaScript Number Methods

[« Previous](#)

[Next Chapter »](#)

Number methods help you to work with numbers.

## Global Methods

JavaScript global functions can be used on all JavaScript data types.

These are the most relevant methods, when working with numbers:

Method	Description
Number()	Returns a number, converted from its argument.
parseFloat()	Parses its argument and returns a floating point number
parseInt()	Parses its argument and returns an integer

# Converting Variables to Numbers

There are 3 JavaScript functions that can be used to convert variables to numbers:

- The `Number()` method
- The `parseInt()` method
- The `parseFloat()` method

These methods are not **number** methods, but **global** JavaScript methods.

---

## The `Number()` Method

**`Number()`**, can be used to convert JavaScript variables to numbers:

# The Number() Method

**Number()**, can be used to convert JavaScript variables to numbers:

## Example

```
x = true;
Number(x);      // returns 1
x = false;
Number(x);      // returns 0
x = new Date();
Number(x);      // returns 1404568027739
x = "10"
Number(x);      // returns 10
x = "10 20"
Number(x);      // returns NaN
```

[Try it Yourself »](#)

# The parseInt() Method

**parseInt()** parses a string and returns a whole number. Spaces are allowed. Only the first number is returned:

## Example

```
parseInt("10");           // returns 10
parseInt("10.33");        // returns 10
parseInt("10 20 30");     // returns 10
parseInt("10 years");     // returns 10
parseInt("years 10");     // returns NaN
```

[Try it yourself »](#)

If the number cannot be converted, NaN (Not a Number) is returned.

# The parseFloat() Method

**parseFloat()** parses a string and returns a number. Spaces are allowed. Only the first number is returned:

## Example

```
parseFloat("10");           // returns 10
parseFloat("10.33");        // returns 10.33
parseFloat("10 20 30");     // returns 10
parseFloat("10 years");     // returns 10
parseFloat("years 10");     // returns NaN
```

Try it yourself »

If the number cannot be converted, NaN (Not a Number) is returned.

# JavaScript Math Object

[« Previous](#)

[Next Chapter »](#)

---

The Math object allows you to perform mathematical tasks on numbers.

---

## The Math Object

The Math object allows you to perform mathematical tasks.

The Math object includes several mathematical methods.

---

One common use of the Math object is to create a random number:

### Example

```
Math.random();    // returns a random number
```



# Math.min() and Math.max()

Math.min() and Math.max() can be used to find the lowest or highest value in a list of arguments:

## Example

```
Math.min(0, 150, 30, 20, -8, -200);    // returns -200
```

Try it yourself »

## Example

```
Math.max(0, 150, 30, 20, -8, -200);    // returns 150
```

Try it yourself »





# Math.round()

Math.round() rounds a number to the nearest integer:

## Example

```
Math.round(4.7);           // returns 5  
Math.round(4.4);           // returns 4
```

Try it Yourself »

# Math.ceil()

Math.ceil() rounds a number **up** to the nearest integer:

## Example

```
Math.ceil(4.4);             // returns 5
```

Try it Yourself »



# Math.floor()

Math.floor() rounds a number **down** to the nearest integer:

## Example

```
Math.floor(4.7);           // returns 4
```

Try it Yourself »

# Math Constants

JavaScript provides 8 mathematical constants that can be accessed with the `Math` object:

## Example

```
Math.E           // returns Euler's number
Math.PI          // returns PI
Math.SQRT2        // returns the square root of 2
Math.SQRT1_2      // returns the square root of 1/2
Math.LN2          // returns the natural logarithm of 2
Math.LN10         // returns the natural logarithm of 10
Math.LOG2E        // returns base 2 logarithm of E
Math.LOG10E       // returns base 10 logarithm of E
```

[Try it yourself »](#)

# Math Object Methods

Method	Description
<code>abs(x)</code>	Returns the absolute value of x
<code>acos(x)</code>	Returns the arccosine of x, in radians
<code>asin(x)</code>	Returns the arcsine of x, in radians
<code>atan(x)</code>	Returns the arctangent of x as a numeric value between $-\pi/2$ and $\pi/2$ radians
<code>atan2(y,x)</code>	Returns the arctangent of the quotient of its arguments
<code>ceil(x)</code>	Returns x, rounded upwards to the nearest integer
<code>cos(x)</code>	Returns the cosine of x (x is in radians)
<code>exp(x)</code>	Returns the value of $E^x$
<code>floor(x)</code>	Returns x, rounded downwards to the nearest integer
<code>log(x)</code>	Returns the natural logarithm (base E) of x
<code>max(x,y,z,...,n)</code>	Returns the number with the highest value
<code>min(x,y,z,...,n)</code>	Returns the number with the lowest value



# What is an Array?

An array is a special variable, which can hold more than one value at a time.

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
var car1 = "Saab";  
var car2 = "Volvo";  
var car3 = "BMW";
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

## Creating an Array

Using an array literal is the easiest way to create a JavaScript Array.

Syntax:

```
var array-name = [item1, item2, ...];
```



# Using the JavaScript Keyword new

The following example also creates an Array, and assigns values to it:

## Example

```
var cars = new Array("Saab", "Volvo", "BMW");
```

[Try it Yourself »](#)



The two examples above do exactly the same. There is no need to use `new Array()`. For simplicity, readability and execution speed, use the first one (the array literal method).

# Arrays are Objects

Arrays are a special type of objects. The **typeof** operator in JavaScript returns "object" for arrays.

But, JavaScript arrays are best described as arrays.

Arrays use **numbers** to access its "elements". In this example, `person[0]` returns John:

## Array:

```
var person = ["John", "Doe", 46];
```

Try it Yourself »

Objects use **names** to access its "members". In this example, `person.firstName` returns John:

## Object:

```
var person = {firstName:"John", lastName:"Doe", age:46};
```



# Array Properties and Methods

The real strength of JavaScript arrays are the built-in array properties and methods:

## Examples

```
var x = cars.length;           // The length property returns the number of elements in cars
var y = cars.sort();           // The sort() method sort cars in alphabetical order
```

Array methods are covered in the next chapter.



# Adding Array Elements

The easiest way to add a new element to an array is using the push method:

## Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.push("Lemon");           // adds a new element (Lemon) to fruits
```

Try it Yourself »

# Associative Arrays

Many programming languages support arrays with named indexes.

Arrays with named indexes are called associative arrays (or hashes).

JavaScript does **not** support arrays with named indexes.

In JavaScript, **arrays** always use **numbered indexes**.

## Example

```
var person = [];  
person[0] = "John";  
person[1] = "Doe";  
person[2] = 46;  
var x = person.length;           // person.length will return 3  
var y = person[0];               // person[0] will return "John"
```

Try it Yourself »

### WARNING !!



If you use a named index, JavaScript will redefine the array to a standard object. After that, **all array methods and properties will produce incorrect results.**



# The Difference Between Arrays and Objects

In JavaScript, **arrays** use **numbered indexes**.

In JavaScript, **objects** use **named indexes**.



Arrays are a special kind of objects, with numbered indexes.

## When to Use Arrays. When to use Objects.

- JavaScript does not support associative arrays.
- You should use **objects** when you want the element names to be **strings (text)**.
- You should use **arrays** when you want the element names to be **numbers**.

## Avoid new Array()

There is no need to use the JavaScript's built-in array constructor **new Array()**.

**Use [] instead.**

These two different statements both create a new empty array named points:

```
var points = new Array();    // Bad
var points = [];             // Good
```

# JavaScript Array Methods

[« Previous](#)

[Next Chapter »](#)

The strength of JavaScript arrays lies in the array methods.

## Converting Arrays to Strings

The JavaScript method **toString()** converts an array to a string of (comma separated) array values.

### Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
document.getElementById("demo").innerHTML = fruits.toString();
```

### Result

```
Banana,Orange,Apple,Mango
```



# Popping and Pushing

When you work with arrays, it is easy to remove elements and add new elements.

This is what popping and pushing is:

Popping items **out** of an array, or pushing items **into** an array.

---

## Popping

The **pop()** method removes the last element from an array:

### Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.pop();           // Removes the last element ("Mango") from fruits
```

Try it Yourself »

# Pushing

The **push()** method adds a new element to an array (at the end):

## Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.push("Kiwi");           // Adds a new element ("Kiwi") to fruits
```

Try it Yourself »

# Shifting Elements

Shifting is equivalent to popping, working on the first element instead of the last.

The **shift()** method removes the first array element and "shifts" all other elements to a lower index.

## Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.shift();           // Removes the first element "Banana" from fruits
```

Try it Yourself »

The **unshift()** method adds a new element to an array (at the beginning), and "unshifts" older elements:


## Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.unshift("Lemon"); // Adds a new element "Lemon" to fruits
```



# Changing Elements

Array elements are accessed using their **index number**:

 Array **indexes** start with 0. [0] is the first array element, [1] is the second, [2] is the third ....

## Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits[0] = "Kiwi";           // Changes the first element of fruits to "Kiwi"
```

Try it Yourself »



# Deleting Elements

Since JavaScript arrays are objects, elements can be deleted by using the JavaScript operator **delete**:

## Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
delete fruits[0];           // Changes the first element in fruits to undefined
```

Try it Yourself »



Using **delete** on array elements leaves undefined holes in the array. Use `pop()` or `shift()` instead.



# Sorting an Array

The **sort()** method sorts an array alphabetically:

## Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.sort();           // Sorts the elements of fruits
```

Try it Yourself »

# Reversing an Array

The **reverse()** method reverses the elements in an array.

You can use it to sort an array in descending order:

## Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.sort();           // Sorts the elements of fruits  
fruits.reverse();        // Reverses the order of the elements
```



# Joining Arrays

The **concat()** method creates a new array by concatenating two arrays:

## Example

```
var myGirls = ["Cecilie", "Lone"];  
var myBoys = ["Emil", "Tobias", "Linus"];  
var myChildren = myGirls.concat(myBoys);    // Concatenates (joins) myGirls and myBoys
```

[Try it Yourself »](#)

# Slicing an Array

The **slice()** method slices out a piece of an array into a new array.

This example slices out a part of an array starting from array element 1 ("Orange"):

## Example

```
var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
var citrus = fruits.slice(1);
```

Try it Yourself »



Array **indexes** start with 0. [0] is the first array element, [1] is the second, [2] is the third ....



This example slices out a part of an array starting from array element 3 ("Apple"):

## Example

```
var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
var citrus = fruits.slice(3);
```

[Try it Yourself »](#)

The slice() method can take two arguments like slice(1,3).

The method then selects elements from the start argument, and up to (but not including) the end argument.

## Example

```
var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
var citrus = fruits.slice(1, 3);
```

[Try it Yourself »](#)



If the end argument is omitted, like in the first examples, the slice() method slices out the rest of the array.

# JavaScript Dates

[« Previous](#)

[Next Chapter »](#)

The Date object lets you work with dates (years, months, days, hours, minutes, seconds, and milliseconds)

## JavaScript Date Formats

A JavaScript date can be written as a string:

**Wed Feb 22 2023 06:40:29 GMT+0100 (West Africa Standard Time)**

or as a number:

**1677044429322**

Dates written as numbers, specifies the number of milliseconds since January 1, 1970, 00:00:00.

# Displaying Dates

In this tutorial we use a script to display dates inside a `<p>` element with `id="demo"`:

## Example


```
<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = Date();
</script>
```

Try it Yourself »

The script above says: assign the value of `Date()` to the content (`innerHTML`) of the element with `id="demo"`.




 JavaScript counts months from 0 to 11. January is 0. December is 11.

## Date Methods

When a Date object is created, a number of **methods** allow you to operate on it.

Date methods allow you to get and set the year, month, day, hour, minute, second, and millisecond of objects, using either local time or UTC (universal, or GMT) time.

 Date methods are covered in a later chapter.



# Time Zones

When setting a date, without specifying the time zone, JavaScript will use the browser's time zone.

When getting a date, without specifying the time zone, the result is converted to the browser's time zone.

In other words: If a date/time is created in GMT (Greenwich Mean Time), the date/time will be converted to CDT (Central US Daylight Time) if a user browses from central US.

# JavaScript Date Formats

[« Previous](#)

[Next Chapter »](#)


## JavaScript Date Input

There are generally 4 types of JavaScript date formats:

- ISO Dates
- Long Dates
- Short Dates
- Full Format

## JavaScript Date Output

Independent of input format, JavaScript will (by default) output dates in full text string format:



```
Wed Mar 25 2015 01:00:00 GMT+0100 (W. Europe Standard Time)
```

# JavaScript ISO Dates

ISO 8601 is the international standard for the representation of dates and times.

The ISO 8601 syntax (YYYY-MM-DD) is also the preferred JavaScript date format:

## Example (Complete date)

```
var d = new Date("2015-03-25");
```

[Try it Yourself »](#)

It can be written without specifying the day (YYYY-MM):

## Example (Year and month)

```
var d = new Date("2015-03");
```

# JavaScript Long Dates.

Long dates are most often written with a "MMM DD YYYY" syntax like this:

## Example

```
var d = new Date("Mar 25 2015");
```

[Try it Yourself »](#)

Month and day can be in any order:

## Example

```
var d = new Date("25 Mar 2015");
```

And, month can be written in full (January), or abbreviated (Jan):

## Example

```
var d = new Date("January 25 2015");
```

[Try it Yourself »](#)

## Example

```
var d = new Date("Jan 25 2015");
```

[Try it Yourself »](#)

Commas are ignored. Names are case insensitive:

# JavaScript Short Dates.

Short dates are most often written with an "MM/DD/YYYY" syntax like this:

## Example

```
var d = new Date("03/25/2015");
```

[Try it Yourself »](#)

JavaScript will also accept "YYYY/MM/DD":

## Example

```
var d = new Date("2015/03/25");
```

[Try it Yourself »](#)

Month is written before day in all short date and ISO date formats.

» Developers

# Time Zones

JavaScript accepts these time zones:

Time Zone	Description
UTC	Coordinated Universal Time
GMT	Greenwich Mean Time
EDT	(US) Eastern Daylight Time
CDT	(US) Central Daylight Time
MDT	(US) Mountain Daylight Time
PDT	(US) Pacific Daylight Time
EST	(US) Eastern Standard Time
CST	(US) Central Standard Time
MST	(US) Mountain Standard Time
PST	(US) Pacific Standard Time

When setting a date, without specifying the time zone, JavaScript will use the browser's time zone.

Date methods let you get and set date values (years, months, days, hours, minutes, seconds, milliseconds)

## Date Get Methods

Get methods are used for getting a part of a date. Here are the most common (alphabetically):

Method	Description
getDate()	Get the day as a number (1-31)
getDay()	Get the weekday as a number (0-6)
getFullYear()	Get the four digit year (yyyy)
getHours()	Get the hour (0-23)
getMilliseconds()	Get the milliseconds (0-999)
getMinutes()	Get the minutes (0-59)
getMonth()	Get the month (0-11)
getSeconds()	Get the seconds (0-59)
getTime()	Get the time (milliseconds since January 1, 1970)





# The getFullYear() Method

**getFullYear()** returns the year of a date as a four digit number:

## Example

```
<script>
var d = new Date();
document.getElementById("demo").innerHTML = d.getFullYear();
</script>
```

[Try it Yourself »](#)

# The `getDay()` Method

**`getDay()`** returns the weekday as a number (0-6):

## Example

```
<script>
var d = new Date();
document.getElementById("demo").innerHTML = d.getDay();
</script>
```

Try it Yourself »



In JavaScript, the first of the week (0) means "Sunday", even if some countries in the world consider the first day of the week to be "Monday".

# JavaScript If...Else Statements

[« Previous](#)

[Next Chapter »](#)

---

Conditional statements are used to perform different actions based on different conditions.

---

## Conditional Statements

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

- Use **if** to specify a block of code to be executed, if a specified condition is true
  - Use **else** to specify a block of code to be executed, if the same condition is false
  - Use **else if** to specify a new condition to test, if the first condition is false
  - Use **switch** to specify many alternative blocks of code to be executed
-

# The if Statement

Use the **if** statement to specify a block of JavaScript code to be executed if a condition is true.

## Syntax

```
if (condition) {  
    block of code to be executed if the condition is true  
}
```



Note that **if** is in lowercase letters. Uppercase letters (If or IF) will generate a JavaScript error.

## Example

Make a "Good day" greeting if the hour is less than 18:00:

```
if (hour < 18) {  
    greeting = "Good day";  
}
```

The result of greeting will be:

Good day



# The else Statement

Use the **else** statement to specify a block of code to be executed if the condition is false.

```
if (condition) {  
    block of code to be executed if the condition is true  
} else {  
    block of code to be executed if the condition is false  
}
```

## Example

If the hour is less than 18, create a "Good day" greeting, otherwise "Good evening":

```
if (hour < 18) {  
    greeting = "Good day";  
} else {  
    greeting = "Good evening";  
}
```

The result of greeting will be:

Good day

# The else if Statement

Use the **else if** statement to specify a new condition if the first condition is false.

## Syntax

```
if (condition1) {  
    block of code to be executed if condition1 is true  
} else if (condition2) {  
    block of code to be executed if the condition1 is false and condition2 is true  
} else {  
    block of code to be executed if the condition1 is false and condition2 is false  
}
```

## Example

If time is less than 10:00, create a "Good morning" greeting, if not, but time is less than 20:00, create a "Good day" greeting, otherwise a "Good evening":

```
if (time < 10) {  
    greeting = "Good morning";  
} else if (time < 20) {  
    greeting = "Good day";  
} else {  
    greeting = "Good evening";  
}
```

The result of greeting will be:

Good morning

# JavaScript For Loop

[« Previous](#)

[Next Chapter »](#)

Loops can execute a block of code a number of times.

## JavaScript Loops

Loops are handy, if you want to run the same code over and over again, each time with a different value.

Often this is the case when working with arrays:

Instead of writing:

```
text += cars[0] + "<br>";  
text += cars[1] + "<br>";  
text += cars[2] + "<br>";  
text += cars[3] + "<br>";  
text += cars[4] + "<br>";  
text += cars[5] + "<br>";
```





You can write:

```
for (i = 0; i < cars.length; i++) {  
    text += cars[i] + "<br>";  
}
```

[Try it Yourself »](#)

# Different Kinds of Loops

JavaScript supports different kinds of loops:

- **for** - loops through a block of code a number of times
- **for/in** - loops through the properties of an object
- **while** - loops through a block of code while a specified condition is true
- **do/while** - also loops through a block of code while a specified condition is true

## The For Loop

The for loop is often the tool you will use when you want to create a loop.

The for loop has the following syntax:

```
for (statement 1; statement 2; statement 3) {  
    code block to be executed  
}
```

**Statement 1** is executed before the loop (the code block) starts.

**Statement 2** defines the condition for running the loop (the code block).

**Statement 3** is executed each time after the loop (the code block) has been executed.



## Example

```
for (i = 0; i < 5; i++) {  
    text += "The number is " + i + "<br>";  
}
```

[Try it yourself »](#)

From the example above, you can read:

Statement 1 sets a variable before the loop starts (var i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5).

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

# JavaScript While Loop

[« Previous](#)

[Next Chapter »](#)

Loops can execute a block of code as long as a specified condition is true.

## The While Loop

The while loop loops through a block of code as long as a specified condition is true.

### Syntax

```
while (condition) {  
    code block to be executed  
}
```

### Example

## Example

In the following example, the code in the loop will run, over and over again, as long as a variable (i) is less than 10:

### Example

```
while (i < 10) {  
    text += "The number is " + i;  
    i++;  
}
```

Try it yourself »



If you forget to increase the variable used in the condition, the loop will never end. This will crash your browser.





# Question ?